

# `problemsetup`

November 29, 2017  
17:12

## Contents

<b>1</b>	<b>Set up file containing data specific to the current problem</b>	<b>1</b>
<b>2</b>	<b>References</b>	<b>42</b>
<b>3</b>	<b>INDEX</b>	<b>43</b>

## 1 Set up file containing data specific to the current problem

```
$Id: cd8027663092776276b442eb8c7361f0dbd64771 $
```

There are two levels of physics input to DEGAS 2. The input files set up by *datasetup* comprise the reference level: in practice, the sum total of the data available to the code (although in principle it could be smaller). The second level is the problem level of data. This prescribes the species, reactions, materials, and PMI which will serve as the physical model to be used in carrying out the simulation at hand. In some parts of the (internal) code, these are also referred to as the subset data since they represent a subset of the reference data.

At the problem level the concepts of test species and background species are introduced. Most simply, test species are the ones DEGAS 2 will track as they collide off of background species. The use of the word species here is important: both of the test and background lists are subsets of the “species” list. One more precise distinction between the two species types is that we assume that we know the distribution function (in space and velocity) of the background species; in fact, such information is required input to DEGAS 2. On the other hand, we are attempting to *compute* the test species distribution function, moments of which serve as the primary output of DEGAS 2.

The problem input file associated with the symbolic name in `problem_infile` in `degas2.in` consists of five sections, one for each of the problem-level physics subsets, separated by the corresponding heading (in capital letters). It is the object symbols (as opposed to the longer “names”) which are used in these lists. For example,

```
TEST
o D D2 D2+
BACKGROUND
e D+
REACTION
hchex
hionize
h2dis
h2ion
h2dision
h2pdision
h2pdis
h2pdisrec
MATERIALS
mo
mirror
PMI
dreflmo
```

```
hdesorbmo
h2desorbmo
hmirror
h2mirror
```

This file is read by *problemsetup*. The reference lists are searched for these symbols. Associations between the subset lists which are needed by the code for tracking the test species are built up. The required atomic and surface data are read in and repackaged for efficient run-time use. All of the resulting information is written to the netCDF problem file, which corresponds to the symbolic name *problemfile* in *degas2.in*.

```
"problemsetup.f" 1≡
@m FILE 'problemsetup.web'
```

The unnamed module.

```
"problemsetup.f" 1.1≡
program problemsetup
  implicit none_f77
  implicit none_f90
  @#if SUN ∧ FORTRAN77
    call float_abort
  @#endiff
  call readfilenames
  call read_problem
  call init_var0_list
  call init_reaction
  call init_pmi
  call finish_var0_list
  call nc_write_problem
  stop
end
```

⟨ Functions and Subroutines 1.2 ⟩

Read in data from `problem_infile`.

`(Functions and Subroutines 1.2) ≡`

```

subroutine read_problem
  implicit none f77
  pr_common
  rf_common
  implicit none f90

  integer length // Local
  character*LINELEN line
  character*FILELEN tempfile

  (Memory allocation interface 0) // Common
  st_decls

  problem_version = 'unknown'
  tempfile = filenames_arrayproblem_infile
  assert(tempfile ≠ char_undef)
  open(unit = diskin, file = tempfile, form = 'formatted', status = 'old')
  assert(read_string(diskin, line, length))
  assert(length ≤ len(line))
  problem_version = line(: length)

  call init_problem
  if (¬read_string(diskin, line, length)) then
    line = 'END'
  end if

loop1: continue
  if (line ≡ 'END')
    goto eof
  if (line ≡ 'TEST') then
    call read_test(diskin, line)
  else if (line ≡ 'BACKGROUND') then
    call read_background(diskin, line)
  else if (line ≡ 'REACTION') then
    call read_reaction(diskin, line)
  else if (line ≡ 'MATERIALS') then
    call read_materials(diskin, line)
  else if (line ≡ 'PMI') then
    call read_pmi(diskin, line)
  else if (line ≡ 'EXTERNAL_TEST') then
    call read_ex_test(diskin, line)
  else if (line ≡ 'EXTERNALREACTION') then
    call read_ex_reaction(diskin, line)
  end if

  goto loop1

eof: continue
  call check_problem
  return
end

```

See also sections 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 1.10, 1.11, 1.12, 1.13, 1.14, 1.15, 1.16, 1.17, 1.18, 1.19, 1.20, 1.21, 1.22, 1.23, 1.24, 1.25, 1.26, 1.27, and 1.28.

This code is used in section 1.1.

Initialize problem.

⟨ Functions and Subroutines 1.2 ⟩ +≡

```
subroutine init_problem
  implicit none_f77
  pr_common // Common
  sp_common
  ps_common
  implicit none_f90
  integer i // Local

  ⟨ Memory allocation interface 0 ⟩

  call nc_read_elements
  call nc_read_species
  call nc_read_reactions
  call nc_read_materials
  call nc_read_pmi

  pr_test_num = 0
  pr_background_num = 0
  pr_ex_test_num = 0
  pr_reaction_num = 0
  pr_exrc_num = 0
  pr_materials_num = 0
  pr_pmi_num = 0

  num_generics = 0
  var_alloc(generics)
  do i = 1, sp_num
    genericsi = 0
  end do

  return
end
```

Read in test particle labels; make required entries in generic species arrays.

```

⟨ Functions and Subroutines 1.2 ⟩ +≡
  subroutine read_test(unit, line)
    implicit none f77
    pr_common // Common
    sp_common
    ps_common
    implicit none f90
    integer unit // Input
    character(*) line // Output
    integer length, p, b, e, i // Local

    ⟨ Memory allocation interface 0 ⟩
    st_decls

    var_alloc(problem_species_test)
    do i = 1, sp_num
      problem_species_test_i = 0
    end do

loop1: continue
  if (¬read_string(unit, line, length))
    line = 'END'
  if (line ≡ 'END' ∨ line ≡ 'TEST' ∨ line ≡ 'BACKGROUND' ∨ line ≡ 'REACTION' ∨ line ≡
      'MATERIALS' ∨ line ≡ 'PMI' ∨ line ≡ 'EXTERNAL_TEST' ∨ LINE ≡ 'EXTERNALREACTION')
    goto eof
  assert(length ≤ len(line))
  length = parse_string(line(:length))

  p = 0
loop2: continue
  if (¬next_token(line(:length), b, e, p))
    goto loop1
  pr_test_num++
  var_realloca(problem_test.sp)
  pr_test(pr_test_num) = sp_lookup(line(b:e))
  assert(pr_test(pr_test_num) > 0)
  problem_species_test(pr_test(pr_test_num)) = pr_test_num

  if (generics_sp_generic(pr_test(pr_test_num)) ≡ 0) then /* First appearance of this generic species;
    update num_generics and generics. In case this entry in the test list is not the archetype
    generic of its isotopic family (i.e., the value of sp_generic), need to also enter num_generics in
    its slot in the generics array. E.g., if D2 were in a problem, but the archetype generic H2 were
    not. */
    num_generics++
    generics_sp_generic(pr_test(pr_test_num)) = num_generics
  if (pr_test(pr_test_num) ≠ sp_generic(pr_test(pr_test_num)))
    generics_pr_test(pr_test_num) = num_generics

  else /* This entry is equivalent to a generic species which has already been added to the generics
    array. */
    generics_pr_test(pr_test_num) = generics_sp_generic(pr_test(pr_test_num))
  end if

  goto loop2

```

```
eof: return  
end
```

Read in background particle labels; make required entries in generic species arrays.

```

⟨ Functions and Subroutines 1.2 ⟩ +≡
  subroutine read_background(unit, line)
    implicit none f77
    pr_common // Common
    sp_common
    ps_common
    implicit none f90
    integer unit // Input
    character(*) line // Output
    integer length, p, b, e, i // Local
    ⟨ Memory allocation interface 0 ⟩
    st_decls

    var_alloc(problem_species_background)
    do i = 1, sp_num
      problem_species_background_i = 0
    end do

loop1: continue
  if (¬read_string(unit, line, length))
    line = 'END'
  if (line ≡ 'END' ∨ line ≡ 'TEST' ∨ line ≡ 'BACKGROUND' ∨ line ≡ 'REACTION' ∨ line ≡
      'MATERIALS' ∨ line ≡ 'PMI' ∨ line ≡ 'EXTERNAL_TEST' ∨ LINE ≡ 'EXTERNALREACTION')
    goto eof
  assert(length ≤ len(line))
  length = parse_string(line)

  p = 0
loop2: continue
  if (¬next_token(line(:length), b, e, p))
    goto loop1
  pr_background_num++
  var_realloca(problem_background_sp)
  pr_background(pr_background_num) = sp_lookup(line(b:e))
  assert(pr_background(pr_background_num) > 0)
  problem_species_background(pr_background(pr_background_num)) = pr_background_num

  /* Handle generics just as we did for test species: */
  if (generics_sp_generic(pr_background(pr_background_num)) ≡ 0) then
    num_generics++
    generics_sp_generic(pr_background(pr_background_num)) = num_generics
    if (pr_background(pr_background_num) ≠ sp_generic(pr_background(pr_background_num)))
      generics_pr_background(pr_background_num) = num_generics
  else
    generics_pr_background(pr_background_num) = generics_sp_generic(pr_background(pr_background_num))
  end if

  goto loop2

eof: return
end

```

Read in reaction labels.

```

⟨ Functions and Subroutines 1.2 ⟩ +≡
subroutine read_reaction(unit, line)
  implicit none_f77
  pr_common // Common
  rc_common
  implicit none_f90
  integer unit // Input
  character*(*) line // Output
  integer length, p, b, e // Local

  ⟨ Memory allocation interface 0 ⟩
  st_decls

  pr_reaction_dim = 0
  assert(pr_reaction_num ≡ 0)

loop1: continue
  if (¬read_string(unit, line, length))
    line = 'END'
  if (line ≡ 'END' ∨ line ≡ 'TEST' ∨ line ≡ 'BACKGROUND' ∨ line ≡ 'REACTION' ∨ line ≡
      'MATERIALS' ∨ line ≡ 'PMI' ∨ line ≡ 'EXTERNAL_TEST' ∨ line ≡ 'EXTERNALREACTION')
    goto eof
  assert(length ≤ len(line))
  length = parse_string(line)

  p = 0
loop2: continue
  if (¬next_token(line(:length), b, e, p))
    goto loop1
  pr_reaction_dim++
  var_realloca(problem_rc)
  pr_reaction(pr_reaction_dim) = rc_lookup(line(b:e))
  assert(pr_reaction(pr_reaction_dim) > 0)
  goto loop2

eof: continue
  pr_reaction_num = pr_reaction_dim
  if (pr_reaction_dim ≡ 0) then
    pr_reaction_dim = 1
    var_realloca(problem_rc)
  end if
  return
end

```

Read in materials labels.

```

⟨ Functions and Subroutines 1.2 ⟩ +≡
subroutine read_materials(unit, line)
  implicit none_f77
  pr_common // Common
  ma_common
  implicit none_f90
  integer unit // Input
  character(*) line // Output
  integer length, p, b, e, i // Local

  ⟨ Memory allocation interface 0 ⟩
  st_decls

  var_alloc(problem_materials_sub)
  do i = 1, ma_num
    problem_materials_sub(i) = 0
  end do

loop1: continue
  if (¬read_string(unit, line, length))
    line = 'END'
  if (line ≡ 'END' ∨ line ≡ 'TEST' ∨ line ≡ 'BACKGROUND' ∨ line ≡ 'REACTION' ∨ line ≡
      'MATERIALS' ∨ line ≡ 'PMI' ∨ line ≡ 'EXTERNAL_TEST' ∨ line ≡ 'EXTERNAL_REACTION')
    goto eof
  assert(length ≤ len(line))
  length = parse_string(line)

  p = 0
loop2: continue
  if (¬next_token(line(:length), b, e, p))
    goto loop1
  pr_materials_num ++
  var_realloc(problem_materials_ref)
  pr_mat_ref(pr_materials_num) = ma_lookup(line(b:e))
  assert(pr_mat_ref(pr_materials_num) > 0)
  problem_materials_sub(pr_mat_ref(pr_materials_num)) = pr_materials_num

  goto loop2

eof: return
end

```

Read in plasma-materials interaction labels.

```

⟨ Functions and Subroutines 1.2 ⟩ +≡
  subroutine read_pmi(unit, line)
    implicit none_f77
    pr_common // Common
    pm_common
    implicit none_f90
    integer unit // Input
    character(*) line // Output
    integer length, p, b, e, i // Local

    ⟨ Memory allocation interface 0 ⟩
    st_decls

    var_alloc(problem_pmi_sub)
    do i = 1, pm_num
      problem_pmi_sub(i) = 0
    end do

loop1: continue
  if (¬read_string(unit, line, length))
    line = 'END'
  if (line ≡ 'END' ∨ line ≡ 'TEST' ∨ line ≡ 'BACKGROUND' ∨ line ≡ 'REACTION' ∨ line ≡
      'MATERIALS' ∨ line ≡ 'PMI' ∨ line ≡ 'EXTERNAL_TEST' ∨ line ≡ 'EXTERNAL_reaction')
    goto eof
  assert(length ≤ len(line))
  length = parse_string(line)

  p = 0
loop2: continue
  if (¬next_token(line(:length), b, e, p))
    goto loop1
  pr_pmi_num++
  var_realloca(problem_pmi_ref)
  pr_pm_ref(pr_pmi_num) = pm_lookup(line(b:e))
  assert(pr_pm_ref(pr_pmi_num) > 0)
  problem_pmi_sub(pr_pm_ref(pr_pmi_num)) = pr_pmi_num

  goto loop2

eof: return
end

```

Read in external test particle labels.

```

⟨ Functions and Subroutines 1.2 ⟩ +≡
  subroutine read_ex_test(unit, line)
    implicit none f77
    pr_common // Common
    sp_common
    implicit none f90
    integer unit // Input
    character*(*) line // Output
    integer length, p, b, e, i // Local

    ⟨ Memory allocation interface 0 ⟩
    st_decls

loop1: continue
  if (¬read_string(unit, line, length))
    line = 'END'
  if (line ≡ 'END' ∨ line ≡ 'TEST' ∨ line ≡ 'BACKGROUND' ∨ line ≡ 'REACTION' ∨ line ≡
      'MATERIALS' ∨ line ≡ 'PMI' ∨ line ≡ 'EXTERNAL_TEST' ∨ line ≡ 'EXTERNAL_reaction')
    goto eof
  assert(length ≤ len(line))
  length = parse_string(line(:length))

  p = 0
loop2: continue
  if (¬next_token(line(:length), b, e, p))
    goto loop1
  pr_ex_test_dim++
  var_realloc(problem_ex_test_sp)
  pr_ex_test(pr_ex_test_dim) = sp_lookup(line(b:e))
  assert(pr_ex_test(pr_ex_test_dim) > 0)

  goto loop2

eof: continue
  pr_ex_test_num = pr_ex_test_dim

  return
end

```

Read in external reaction labels.

```

⟨ Functions and Subroutines 1.2 ⟩ +≡
subroutine read_ex_reaction(unit, line)
  implicit none_f77
  pr_common // Common
  rc_common
  implicit none_f90
  integer unit // Input
  character*(*) line // Output
  integer length, p, b, e // Local

  ⟨ Memory allocation interface 0 ⟩
  st_decls

  pr_exrc_dim = 0
  assert(pr_exrc_num ≡ 0)

loop1: continue
  if (¬read_string(unit, line, length))
    line = 'END'
  if (line ≡ 'END' ∨ line ≡ 'TEST' ∨ line ≡ 'BACKGROUND' ∨ line ≡ 'REACTION' ∨ line ≡
      'MATERIALS' ∨ line ≡ 'PMI' ∨ line ≡ 'EXTERNAL_TEST' ∨ line ≡ 'EXTERNAL_REACTION')
    goto eof
  assert(length ≤ len(line))
  length = parse_string(line)

  p = 0
loop2: continue
  if (¬next_token(line(:length), b, e, p))
    goto loop1
  pr_reaction_dim++
  var_realloca(problem_rc)
  pr_reaction(pr_reaction_dim) = rc_lookup(line(b:e))
  assert(pr_reaction(pr_reaction_dim) > 0)
  pr_exrc_dim++
  var_realloca(problem_external_reaction)
  pr_ex_rc(pr_exrc_dim) = pr_reaction(pr_reaction_dim)
  goto loop2

eof: continue
  pr_reaction_num = pr_reaction_dim
  pr_exrc_num = pr_exrc_dim

  return
end

```

Setup initial entries in list of dependent variables. This routine does two of the three “sections” of this array: hardwired (entries corresponding to macros) and background terms (repeated for each background species). The reaction-specific entries will be set in *xs\_copy* as needed.

```

⟨ Functions and Subroutines 1.2 ⟩ +≡
subroutine init_var0_list
  implicit none_f77
  pr_common    // Common
  sp_common
  implicit none_f90
  integer i    // Local
  ⟨ Memory allocation interface 0 ⟩
  pr_var0_num = 1    // Will force mem_alloc to be called.
  var_alloc(pr_var0_list)
  pr_var0_num = pr_var_mass_change - 1 + pr_num_change_vars * (pr_background_num + pr_test_num + 1)
  // The last “1” here accounts for the generic entries.
  @#if 0
    var_realloc(pr_var0_list)    // Need here since will be using later also
  @#endif
  var_realloc(pr_var0_list, 1, mem_size(pr_var0_num))
  do i = 1, pr_var0_num
    pr_var0_listi = char_uninit
  end do

  pr_var0_listpr_var_unknown = 'unknown'
  pr_var0_listpr_var_mass = 'mass'
  pr_var0_listpr_var_momentum_vector = 'momentum_vector'
  pr_var0_listpr_var_momentum_2 = 'momentum_2'
  pr_var0_listpr_var_momentum_3 = 'momentum_3'
  pr_var0_listpr_var_energy = 'energy'
  pr_var0_listpr_var_angle = 'angle'
  pr_var0_listpr_var_emitter_v_vector = 'emitter_v_vector'
  pr_var0_listpr_var_emitter_v_2 = 'emitter_v_2'
  pr_var0_listpr_var_emitter_v_3 = 'emitter_v_3'
  pr_var0_listpr_var_emitter_vf_Maxwell_vector = 'emitter_vf_Maxwell_vector'
  pr_var0_listpr_var_emitter_vf_Maxwell_2 = 'emitter_vf_Maxwell_2'
  pr_var0_listpr_var_emitter_vf_Maxwell_3 = 'emitter_vf_Maxwell_3'
  pr_var0_listpr_var_emitter_vth_Maxwell = 'emitter_vth_Maxwell'
  pr_var0_listpr_var_xz_stress = 'xz_stress'    // For Couette example

  pr_var0_listpr_var_mass_change = 'mass_change'    // Generic entries
  pr_var0_listpr_var_momentum_change_vector = 'momentum_change_vector'
  pr_var0_listpr_var_momentum_change_2 = 'momentum_change_2'
  pr_var0_listpr_var_momentum_change_3 = 'momentum_change_3'
  pr_var0_listpr_var_energy_change = 'energy_change'
  pr_var0_listpr_var_mass_in = 'mass_in'
  pr_var0_listpr_var_momentum_in_vector = 'momentum_in_vector'
  pr_var0_listpr_var_momentum_in_2 = 'momentum_in_2'
  pr_var0_listpr_var_momentum_in_3 = 'momentum_in_3'
  pr_var0_listpr_var_energy_in = 'energy_in'
  pr_var0_listpr_var_mass_out = 'mass_out'
  pr_var0_listpr_var_momentum_out_vector = 'momentum_out_vector'
  pr_var0_listpr_var_momentum_out_2 = 'momentum_out_2'

```

```

pr-var0-listpr-var-momentum-out_3 = 'momentum_out_3'
pr-var0-listpr-var-energy-out = 'energy_out'

assert(pr-background-num ≥ 1)
do i = 1, pr-background-num
    pr-var0-listpr-var-problem-sp-index(pr-var-mass-change, pr-problem-sp-back(i)) = 'back_mass_change_' ||
        sp-sy(pr-background(i))
    pr-var0-listpr-var-problem-sp-index(pr-var-momentum-change-vector, pr-problem-sp-back(i)) =
        'back_momentum_change_vector_' || sp-sy(pr-background(i))
    pr-var0-listpr-var-problem-sp-index(pr-var-momentum-change_2, pr-problem-sp-back(i)) =
        'back_momentum_change_2_' || sp-sy(pr-background(i))
    pr-var0-listpr-var-problem-sp-index(pr-var-momentum-change_3, pr-problem-sp-back(i)) =
        'back_momentum_change_3_' || sp-sy(pr-background(i))
    pr-var0-listpr-var-problem-sp-index(pr-var-energy-change, pr-problem-sp-back(i)) =
        'back_energy_change_' || sp-sy(pr-background(i))
    pr-var0-listpr-var-problem-sp-index(pr-var-mass-in, pr-problem-sp-back(i)) = 'back_mass_in_' ||
        sp-sy(pr-background(i))
    pr-var0-listpr-var-problem-sp-index(pr-var-momentum-in-vector, pr-problem-sp-back(i)) =
        'back_momentum_in_vector_' || sp-sy(pr-background(i))
    pr-var0-listpr-var-problem-sp-index(pr-var-momentum-in_2, pr-problem-sp-back(i)) =
        'back_momentum_in_2_' || sp-sy(pr-background(i))
    pr-var0-listpr-var-problem-sp-index(pr-var-momentum-in_3, pr-problem-sp-back(i)) =
        'back_momentum_in_3_' || sp-sy(pr-background(i))
    pr-var0-listpr-var-problem-sp-index(pr-var-energy-in, pr-problem-sp-back(i)) = 'back_energy_in_' ||
        sp-sy(pr-background(i))
    pr-var0-listpr-var-problem-sp-index(pr-var-mass-out, pr-problem-sp-back(i)) = 'back_mass_out_' ||
        sp-sy(pr-background(i))
    pr-var0-listpr-var-problem-sp-index(pr-var-momentum-out-vector, pr-problem-sp-back(i)) =
        'back_momentum_out_vector_' || sp-sy(pr-background(i))
    pr-var0-listpr-var-problem-sp-index(pr-var-momentum-out_2, pr-problem-sp-back(i)) =
        'back_momentum_out_2_' || sp-sy(pr-background(i))
    pr-var0-listpr-var-problem-sp-index(pr-var-momentum-out_3, pr-problem-sp-back(i)) =
        'back_momentum_out_3_' || sp-sy(pr-background(i))
    pr-var0-listpr-var-problem-sp-index(pr-var-energy-out, pr-problem-sp-back(i)) = 'back_energy_out_' ||
        sp-sy(pr-background(i))
end do
assert(pr-test-num ≥ 1)
do i = 1, pr-test-num
    pr-var0-listpr-var-problem-sp-index(pr-var-mass-change, pr-problem-sp-test(i)) = 'test_mass_change_' ||
        sp-sy(pr-test(i))
    pr-var0-listpr-var-problem-sp-index(pr-var-momentum-change-vector, pr-problem-sp-test(i)) =
        'test_momentum_change_vector_' || sp-sy(pr-test(i))
    pr-var0-listpr-var-problem-sp-index(pr-var-momentum-change_2, pr-problem-sp-test(i)) =
        'test_momentum_change_2_' || sp-sy(pr-test(i))
    pr-var0-listpr-var-problem-sp-index(pr-var-momentum-change_3, pr-problem-sp-test(i)) =
        'test_momentum_change_3_' || sp-sy(pr-test(i))
    pr-var0-listpr-var-problem-sp-index(pr-var-energy-change, pr-problem-sp-test(i)) =
        'test_energy_change_' || sp-sy(pr-test(i))
    pr-var0-listpr-var-problem-sp-index(pr-var-mass-in, pr-problem-sp-test(i)) = 'test_mass_in_' ||
        sp-sy(pr-test(i))
    pr-var0-listpr-var-problem-sp-index(pr-var-momentum-in-vector, pr-problem-sp-test(i)) =
        'test_momentum_in_vector_' || sp-sy(pr-test(i))
    pr-var0-listpr-var-problem-sp-index(pr-var-momentum-in_2, pr-problem-sp-test(i)) =

```

```

'test_momentum_in_2_ || sp_sy(pr_test(i))
pr_var0_listpr_var_problem_sp_index(pr_var_momentum_in_3, pr_problem_sp_test(i)) =
'test_momentum_in_3_ || sp_sy(pr_test(i))
pr_var0_listpr_var_problem_sp_index(pr_var_energy_in, pr_problem_sp_test(i)) = 'test_energy_in_ ||
sp_sy(pr_test(i))
pr_var0_listpr_var_problem_sp_index(pr_var_mass_out, pr_problem_sp_test(i)) = 'test_mass_out_ ||
sp_sy(pr_test(i))
pr_var0_listpr_var_problem_sp_index(pr_var_momentum_out_vector, pr_problem_sp_test(i)) =
'test_momentum_out_vector_ || sp_sy(pr_test(i))
pr_var0_listpr_var_problem_sp_index(pr_var_momentum_out_2, pr_problem_sp_test(i)) =
'test_momentum_out_2_ || sp_sy(pr_test(i))
pr_var0_listpr_var_problem_sp_index(pr_var_momentum_out_3, pr_problem_sp_test(i)) =
'test_momentum_out_3_ || sp_sy(pr_test(i))
pr_var0_listpr_var_problem_sp_index(pr_var_energy_out, pr_problem_sp_test(i)) = 'test_energy_out_ ||
sp_sy(pr_test(i))
end do
return
end

```

Finish assignments to the list of dependent variable labels. This routine will serve to make an additional assignments to *pr\_var0\_list* beyond those made by *xs\_copy* and to clean up its memory allocation.

The only task carried out here at present is to set up additional emission rate variables corresponding to each wavelength. Here's the long-winded explanation of why this is needed: "In general" the emission rate and wavelength variables should both have the name of the line appended to the variable names. For the case of greatest interest, this would entail carrying around 3 sets of (usually) identical data - one for each hydrogen isotope. Alternatively, the problem is that we're saying "all of the atomic physics for the hydrogen isotopes is the same except the wavelength"; the fact that there is an exception strictly speaking invalidates the notion of generic species. However, the other payoffs of generic species heavily outweigh this one exception. Here, the data files carry a generic name and multiple wavelengths. Each of these should now be in the *pr\_var0\_list*. However, the scoring dependent variables will refer to emission rates for explicit lines. We define here those additional emission rates, one for each wavelength. The individual reaction-processing subroutines will need to make the connection between the generic emission rate variable and the specific one based on the computed value of the wavelength. If needed, this procedure could be easily extended to handle other wavelength-dependent data.

```

⟨ Functions and Subroutines 1.2 ⟩ +≡
subroutine finish_var0_list
  implicit none_f77
  pr_common
  implicit none_f90
  integer i, i_e_rate, num_lines, old_num      // Local
  character*pr_tag_string_length line, lines_pr_max_lines
  st_decls
  ⟨ Memory allocation interface 0 ⟩
  num_lines = 0
  old_num = pr_var0_num
  do i = 1, pr_var0_num
    if (pr_var0_list_i SP(1 : 10) ≡ 'wavelength') then
      line = pr_var0_list_i SP(11 :)
      i_e_rate = string_lookup('emission_rate' || line, pr_var0_list, pr_var0_num)
      if (i_e_rate ≡ 0) then    // Add to list if not there already
        num_lines ++
        pr_var0_num ++
        var_realloc(pr_var0_list)
        lines_num_lines = line
      end if
    end if
  end do
  if (num_lines > 0) then
    #if 0
      var_realloc(pr_var0_list, pr_var0_num, pr_var0_num + num_lines)
    #endif
    #if 0
      pr_var0_num += num_lines
      var_realloc(pr_var0_list)
    #endif
    do i = 1, num_lines
      pr_var0_list_{old_num+i} = 'emission_rate' || lines_i
  end if
end subroutine

```

```
    end do
end if
var_reallocb(pr_var0_list)
return
end
```

Check problem specification. This routine sets up the arrays which will actually be used to implement reactions and PMI in the run. This involves not only matching up reactions and PMI with the test species, but also allowing for the isotopically equivalent combinations.

```

⟨ Functions and Subroutines 1.2 ⟩ +≡
subroutine check_problem
  implicit none_f77
  pr_common // Common
  sp_common
  rc_common
  ma_common
  pm_common
  ps_common
  implicit none_f90
  integer i, j, k, l, m, ts_reagent, bk_reagent // Local
  pr_test_decl(test)
  pr_background_decl(background)
  pr_generic_decl(generic)
  pr_generic_decl(generic_reagentspr_bkrc_reagent_max)
  pr_background_decl(backpr_bkrc_reagent_max)
  sp_decl(reagentsrc_reagent_max)
  ⟨ Memory allocation interface 0 ⟩
  st_decls

  var_reallocb(problem_test_sp)
  var_reallocb(problem_background_sp)
  var_reallocb(problem_rc)
  var_reallocb(problem_materials_ref)
  var_reallocb(problem_pmi_ref)
  if (pr_ex_test_num ≡ 0) then
    pr_ex_test_dim = 1
    var_alloc(problem_ex_test_sp)
    pr_ex_test(pr_ex_test_dim) = int_unused
  else
    assert(pr_exrc_num > 0)
  end if
  if (pr_exrc_num ≡ 0) then
    pr_exrc_dim = 1
    var_alloc(problem_external_reaction)
    pr_ex_rc(pr_exrc_dim) = int_unused
  else
    assert(pr_ex_test_num > 0)
  end if

  write(stdout, *) 'Number_of:'
  write(stdout, *) ' test_species= ', pr_test_num
  write(stdout, *) ' background_species= ', pr_background_num
  write(stdout, *) ' materials= ', pr_materials_num
  write(stdout, *) ' reactions= ', pr_reaction_num
  write(stdout, *) ' plasma-material_interactions= ', pr_pmi_num
  if (pr_ex_test_num > 0)
    write(stdout, *) ' external_test_species= ', pr_ex_test_num
  if (pr_exrc_num > 0)

```

```

    write (stdout, *) 'External_reactions=', pr_exrc_num
    write (stdout, *) 'Test_species_are:'
do i = 1, pr_test_num
    write (stdout, *) pr_test(i), ',', trim(sp_name(pr_test(i)))
end do
write (stdout, *) 'Background_species_are:'
do i = 1, pr_background_num
    write (stdout, *) pr_background(i), ',', trim(sp_name(pr_background(i)))
end do
write (stdout, *) 'Reference_material_indices_are:'
do i = 1, pr_materials_num
    write (stdout, *) pr_mat_ref(i), ',', trim(ma_name(pr_mat_ref(i)))
end do
write (stdout, *) 'Reference_reaction_indices_are:'
if (pr_reaction_num > 0) then
    do i = 1, pr_reaction_num
        write (stdout, *) pr_reaction(i), ',', trim(rc_name(pr_reaction(i)))
    end do
end if
write (stdout, *) 'Reference_PMI_indices_are:'
do i = 1, pr_pmi_num
    write (stdout, *) pr_pm_ref(i), ',', trim(pm_name(pr_pm_ref(i)))
end do
if (pr_ex_test_num > 0) then
    write (stdout, *) 'External_test_species_are:'
    do i = 1, pr_ex_test_num
        write (stdout, *) pr_ex_test(i), ',', trim(sp_name(pr_ex_test(i)))
    end do
end if
if (pr_exrc_num > 0) then
    write (stdout, *) 'External_reactions_are:'
    do i = 1, pr_exrc_num
        write (stdout, *) pr_ex_rc(i), ',', trim(rc_name(pr_ex_rc(i)))
    end do
end if

var_alloc(problem_reaction_num)
var_alloc(problem_test_reaction)
var_alloc(problem_test_background)
var_alloc(problem_pmi_case_num)
var_alloc(problem_pmi_cases)
var_alloc(problem_pmi_num_arrange)
var_alloc(problem_pmi_prod_mult)
var_alloc(problem_pmi_products)
var_alloc(problem_num_arrangements)
var_alloc(problem_prod_mult)
var_alloc(problem_test_products)
var_alloc(problem_external_test_reaction_num)
var_alloc(problem_external_test_reaction)
var_alloc(problem_external_test_background)
var_alloc(problem_exrc_products)

assert(pr_max_equiv ≥ max(pr_test_num, pr_background_num))

```

```

var_alloc(num_equiv)
var_alloc(equivalents)

do i = 1, pr_test_num
    do j = 1, pr_reaction_max
        pr_ts_rc(i, j) = int_unused
        pr_ts_bk(i, j) = int_unused
        pr_num_arrangements(i, j) = int_unused
        do k = 1, pr_arrangement_max
            pr_prod_mult(i, j, k) = real_unused
            do l = 1, rc_product_max
                pr_ts_prod(i, j, k, l) = int_unused
            end do
        end do
    end do
end do

do i = 1, pr_max_equiv
    do j = 1, num_generics
        num_equiv_j = 0
        equivalents_i, j = 0
    end do
end do

do i = 1, pr_ex_test_dim
    pr_ex_ts_rc_num(i) = int_unused
    do j = 1, pr_reaction_max
        pr_ex_ts_rc(i, j) = int_unused
        pr_ex_ts_bk(i, j) = int_unused
        do k = 1, rc_product_max
            pr_ex_rc_prod(i, j, k) = int_unused
        end do
    end do
end do

end do /* For each test species, first obtain the index of the corresponding generic archetype,
generic, increment the number of equivalents to this archetype, and store the species index of
this test in the equivalents array. */

do i = 1, pr_test_num
    generic = generic(pr_test(i))
    num_equiv_generic++
    equivalents_num_equiv_generic, generic = pr_test(i)
end do

/* Do same for generics of background species. Since the same species may be a test, as well as a
background, check to see if it's already on the list (call to int_lookup). */
do i = 1, pr_background_num
    generic = generic(pr_background(i))
    if ((num_equiv_generic == 0) || (int_lookup(pr_background(i), equivalents_1, generic, pr_max_equiv) == 0))
        then
            num_equiv_generic++
            equivalents_num_equiv_generic, generic = pr_background(i)
        endif
    end do

    /* We can only deal with binary collisions currently */
    if (pr_reaction_num > 0) then

```

```

do  $j = 1, pr\_reaction\_num$ 
   $assert(rc\_reagent\_num(pr\_reaction(j)) \equiv 2)$ 
end do
end if

do  $i = 1, pr\_test\_num$ 
   $pr\_rc\_num(i) = 0$ 
end do
 $pr\_bkrc\_dim = 0$ 

if ( $pr\_ex\_test\_num > 0$ ) then
  do  $i = 1, pr\_ex\_test\_num$ 
     $pr\_ex\_ts\_rc\_num(i) = 0$ 
  end do
end if

if ( $pr\_reaction\_num > 0$ ) then
  do  $j = 1, pr\_reaction\_num$ 
    if ( $int\_lookup(pr\_reaction(j), problem\_external\_reaction, pr\_exrc\_dim) \equiv 0$ ) then
      /* The original version of this code permitted an arbitrary ordering of test and background in reagent specification (restricted to binary collisions). With the introduction of neutral-neutral elastic scattering reactions (reagents having the same species), a specific ordering had to be chosen. The convention used in many outside data sources (e.g., Aladdin) is consistent with “background + test”. So, this ordering is chosen here as well. Note: assuming that we do not need code here to deal with species-specific reactions (asserts to follow should catch exceptions). */
       $ts\_reagent = 0$ 
      do  $i = 1, pr\_bkrc\_reagent\_max$ 
         $generic\_reagents_i = 0$ 
      end do
      do  $i = 1, rc\_reagent\_num(pr\_reaction(j))$ 
         $generic\_reagents_i = generic\_reagents_{rc\_reagent(pr\_reaction(j), i)}$ 
        if ( $pr\_test\_lookup(equivalents_{1, generic\_reagents_i}) \neq 0$ ) then
           $ts\_reagent = 2$  // else: is a background reaction (below)
           $bk\_reagent = 1$ 
        end if
      end do
      /* Done setting generic_reagents; use to assign remaining pr arrays */
    if ( $ts\_reagent > 0$ ) then
      if ( $rc\_gen(pr\_reaction(j)) \equiv rc\_generic\_no$ ) then
         $test = pr\_test.lookup(rc\_reagent(pr\_reaction(j), ts\_reagent))$ 
         $background = pr\_background.lookup(rc\_reagent(pr\_reaction(j), bk\_reagent))$ 
         $assert(pr\_test\_check(test) \wedge pr\_background\_check(background))$ 
         $pr\_rc\_num(test)++$ 
         $assert(pr\_rc\_num(test) \leq pr\_reaction\_max)$ 
         $pr\_ts\_rc(test, pr\_rc\_num(test)) = j$ 
         $pr\_ts\_bk(test, pr\_rc\_num(test)) = background$ 
         $pr\_num\_arrangements(test, pr\_rc\_num(test)) = 1$ 
         $pr\_prod\_mult(test, pr\_rc\_num(test), 1) = one$ 
        do  $k = 1, rc\_product\_num(pr\_reaction(j))$ 
           $pr\_ts\_prod(test, pr\_rc\_num(test), 1, k) = rc\_product(pr\_reaction(j), k)$ 
        end do
      else if ( $rc\_gen(pr\_reaction(j)) \equiv rc\_generic\_yes$ ) then

```

```

do  $k = 1, \text{num\_equiv}_{\text{generic\_reagents}_{ts\_reagent}}$ 
   $test = pr\_test\_lookup(\text{equivalents}_k, \text{generic\_reagents}_{ts\_reagent})$ 
  do  $l = 1, \text{num\_equiv}_{\text{generic\_reagents}_{bk\_reagent}}$ 
     $background = pr\_background\_lookup(\text{equivalents}_l, \text{generic\_reagents}_{bk\_reagent})$ 
     $\text{assert}(pr\_test\_check(test) \wedge pr\_background\_check(background))$ 
     $pr\_rc\_num(test)++$ 
     $\text{assert}(pr\_rc\_num(test) \leq pr\_reaction\_max)$ 
     $pr\_ts\_rc(test, pr\_rc\_num(test)) = j$ 
     $pr\_ts\_bk(test, pr\_rc\_num(test)) = background$ 
     $reagents_{bk\_reagent} = pr\_background(background)$ 
     $reagents_{ts\_reagent} = pr\_test(test)$ 
     $\text{call set\_products}(pr\_test\_args(test), sp\_args(reagents), rc\_args(pr\_reaction(j)))$ 
  end do
  end do
else
   $\text{assert}('Illegal\_value\_of\_rc\_gen(j) \equiv \_) \equiv \_')$ 
end if // Handle test reaction generics
else /* Reactions among background species... We have not established a protocol yet for
   dealing with reactions involving two background species. The typical reactions involve
   a test species (given velocity and position) and a background species (described only
   by a temperature, density, and flow velocity). If we're going to treat reactions between
   two background species in the same way, we'll need to specify which behaves more like
   a test particle. Rather than rely on a convention for the ordering of the reagents on
   input, make this decision here based on mass. The obvious motivation is that one of
   the reagents is inevitably an electron in which case i) its temperature determines the
   reaction rate, and ii) we will not likely be interested in it as a test particle. Whereas,
   the other reagent (e.g., an ion about to be recombined) will be sampled in preparation
   for determining the velocity of the sourced test species. */
if ( $rc\_gen(pr\_reaction(j)) \equiv rc\_generic\_no$ ) then
  do  $k = 1, pr\_bkrc\_reagent\_max$ 
     $back_k = pr\_background\_lookup(rc\_reagent(pr\_reaction(j), k))$ 
     $\text{assert}(pr\_background\_check(back_k))$ 
  end do
   $pr\_bkrc\_dim++$ 
   $var\_realloc(problem\_background\_reaction)$ 
   $var\_realloc(problem\_bkrc\_reagents)$ 
   $var\_realloc(problem\_bkrc\_products)$ 
   $pr\_bk\_rc(pr\_bkrc\_dim) = j$ 
  if ( $sp\_m(pr\_background(back_2)) < sp\_m(pr\_background(back_1))$ ) then
     $pr\_bkrc\_rg(pr\_bkrc\_dim, 1) = back_2$ 
     $pr\_bkrc\_rg(pr\_bkrc\_dim, 2) = back_1$ 
  else
     $pr\_bkrc\_rg(pr\_bkrc\_dim, 1) = back_1$ 
     $pr\_bkrc\_rg(pr\_bkrc\_dim, 2) = back_2$ 
  end if
   $\text{assert}(rc\_product\_num(pr\_reaction(j)) \equiv 1)$ 
  // Otherwise, we need to add machinery to deal with arrangements, etc.
   $pr\_bkrc\_prod(pr\_bkrc\_dim, 1) = rc\_product(pr\_reaction(j), 1)$ 
else if ( $rc\_gen(pr\_reaction(j)) \equiv rc\_generic\_yes$ ) then
  do  $k = 1, \text{num\_equiv}_{\text{generic\_reagents}_1}$ 
     $back_1 = pr\_background\_lookup(\text{equivalents}_k, \text{generic\_reagents}_1)$ 

```

```

        assert(pr_background_check(back_1))
        do l = 1, num_equiv generic_reagents pr_bkrc_reagent_max
            back_pr_bkrc_reagent_max =
                pr_background_lookup(equivalents_l, generic_reagents pr_bkrc_reagent_max)
            assert(pr_background_check(back_pr_bkrc_reagent_max))
            pr_bkrc_dim++
            var_realloca(problem_background_reaction)
            var_realloca(problem_bkrc_reagents)
            var_realloca(problem_bkrc_products)
            pr_bkrc(pr_bkrc_dim) = j
            do m = 1, pr_bkrc_reagent_max
                if (sp_m(pr_background(back_pr_bkrc_reagent_max)) < sp_m(pr_background(back_1)))
                    then
                        pr_bkrc_rg(pr_bkrc_dim, m) = back_pr_bkrc_reagent_max-m+1
                    else
                        pr_bkrc_rg(pr_bkrc_dim, m) = back_m
                    end if
                    reagents_m = pr_background(back_m)
                end do
                call set_bkrc_products(pr_bkrc_dim, sp_args(reagents))
            end do
        end do
    else
        assert('Illegal value of rc_gen(j)', ' ')
    end if // Handle background generics
end if // Test or background (normal reactions)
else /* External reactions. Skipping the processing of generic species variations in the
       interest of simplicity. Do have one potential example (hrecombine), but implementing a
       work-around to handle it for now. The test and background ordering is the same as for
       ordinary reactions, hence the resetting of ts_reagent and bk_reagent. */
ts_reagent = 2
bk_reagent = 1
test = pr_ex_test_lookup(rc_reagent(pr_reaction(j), ts_reagent))
background = pr_background_lookup(rc_reagent(pr_reaction(j), bk_reagent))
assert(pr_ex_test_check(test) ∧ pr_background_check(background))
pr_ex_ts_rc_num(test)++
assert(pr_ex_ts_rc_num(test) ≤ pr_reaction_max)
pr_ex_ts_rc(test, pr_ex_ts_rc_num(test)) = j
pr_ex_ts_bk(test, pr_ex_ts_rc_num(test)) = background
do k = 1, rc_product_num(pr_reaction(j))
    pr_ex_rc_prod(test, pr_ex_ts_rc_num(test), k) = rc_product(pr_reaction(j), k)
end do
end if // Normal or external reactions
end do // Over problem reactions
end if /* On pr_reaction_num */ Have to introduce pr_bkrc_dim because netCDF allocates space
        for these arrays even if pr_bkrc_num=0. The problem arises when the netCDF file containing
        these variables is read back in. The size is reported to be non-zero, but the dimensioning
        variables says it should be zero. The awkward solution is to use pr_bkrc_dim as the dimension of
        these arrays and pr_bkrc_num as the actual number of reactions. The former is set to 1 if there
        are no background reactions.
pr_bkrc_num = pr_bkrc_dim
if (pr_bkrc_dim ≡ 0) then

```

```

pr_bkrc_dim = 1
var_alloc(problem_background_reaction)
var_alloc(problem_bkrc_reagents)
var_alloc(problem_bkrc_products)
pr_bk_rc(1) = int_unused
do i = 1, pr_bkrc_reagent_max
    pr_bkrc_rg(1, i) = int_unused
end do
do i = 1, rc_product_max
    pr_bkrc_prod(1, i) = int_unused
end do
else
    var_reallocb(problem_background_reaction)
    var_reallocb(problem_bkrc_reagents)
    var_reallocb(problem_bkrc_products)
end if

do i = 1, pr_test_num
    pr_pm_case_num(i) = 0
    do j = 1, pr_pmi_max
        pr_pm_cases(i, j) = int_unused
        pr_pm_num_arrange(i, j) = int_unused
        do k = 1, pr_arrangement_max
            pr_pm_prod_mult(i, j, k) = real_unused
            do l = 1, rc_product_max
                pr_pm_prod(i, j, k, l) = int_unused
            end do
        end do
    end do
end do
end do /* Here is the analogous code for PMI which sets up the list of PMI in which each test
           species participates. Species-specific PMI are dealt with first: */
do j = 1, pr_pmi_num
    if (pm_gen(pr_pm_ref(j)) ≡ pmi_generic_no) then
        test = pr_test_lookup(pm_reagent(pr_pm_ref(j)))
        pr_pm_case_num(test)++
        pr_pm_cases(test, pr_pm_case_num(test)) = j
        /* For species-specific PMI, the products can be set directly: */
        pr_pm_num_arrange(test, pr_pm_case_num(test)) = 1
        pr_pm_prod_mult(test, pr_pm_case_num(test), 1) = one
        do k = 1, pm_product_num(pr_pm_ref(j))
            pr_pm_prod(test, pr_pm_case_num(test), 1, k) = pm_product(pr_pm_ref(j), k)
        end do
    /* For PMI which refer to generic reagents and product species, proceed as above for
       reactions. The process is simplified because (1) there are no generic materials, (2) the
       ordering of test and materials is hardwired. */
    else if (pm_gen(pr_pm_ref(j)) ≡ pmi_generic_yes) then
        generic_reagents_1 = generic_pm_reagent(pr_pm_ref(j))
        do k = 1, num_equiv_generic_reagents_1
            test = pr_test_lookup(equivalents_k, generic_reagents_1)
            assert(pr_test_check(test))
            pr_pm_case_num(test)++
            pr_pm_cases(test, pr_pm_case_num(test)) = j
    end if
end do

```

```
call set_pmi_products(pr_test_args(test), pm_args(pr_pm_ref(j)))
end do
else
  assert('Illegal value of pm_gen(j)' == ' ')
end if
end do
return
end
```

This is a clearing-house routine used to select the product-setting routine specific for each reaction.

```

⟨ Functions and Subroutines 1.2 ⟩ +≡
  subroutine set_products(pr_test_dummy(ts), sp_dummy(reagents), rc_dummy(r))
    implicit none f77
    pr_common
    rc_common
    implicit none f90

    pr_test_decl(ts) // Input
    rc_decl(r)
    sp_decl(reagents_*)
    integer i // Local
    real sum

    if (rc_reaction_type(r) ≡ 'ionize') then
      call set_prod_ionize(pr_test_args(ts), sp_args(reagents), rc_args(r))
    else if (rc_reaction_type(r) ≡ 'ionize_suppress') then
      call set_prod_ionize(pr_test_args(ts), sp_args(reagents), rc_args(r))
    else if (rc_reaction_type(r) ≡ 'chargeX') then
      call set_prod_chargeX(pr_test_args(ts), sp_args(reagents), rc_args(r))
    else if (rc_reaction_type(r) ≡ 'dissoc') then
      call set_prod_dissoc(pr_test_args(ts), sp_args(reagents), rc_args(r))
    else if (rc_reaction_type(r) ≡ 'dissoc_rec') then
      call set_prod_dissoc_rec(pr_test_args(ts), sp_args(reagents), rc_args(r))
    else if (rc_reaction_type(r) ≡ 'dissoc_cramd') then
      call set_prod_dissoc_cramd(pr_test_args(ts), sp_args(reagents), rc_args(r))
    else if (rc_reaction_type(r) ≡ 'elastic') then
      call set_prod_elastic(pr_test_args(ts), sp_args(reagents), rc_args(r))
    else if (rc_reaction_type(r) ≡ 'bgk_elastic') then
      call set_prod_elastic(pr_test_args(ts), sp_args(reagents), rc_args(r))
    else if (rc_reaction_type(r) ≡ 'excitation') then
      call set_prod_excite(pr_test_args(ts), sp_args(reagents), rc_args(r))
    else if (rc_reaction_type(r) ≡ 'deexcitation') then
      call set_prod_deexcite(pr_test_args(ts), sp_args(reagents), rc_args(r))
    else
      pr_num_arrangements(ts, pr_rc_num(ts)) = 0
      return
    end if

    /* Rescale multiplicity so that it becomes a probability */
    sum = zero
    do i = 1, pr_num_arrangements(ts, pr_rc_num(ts))
      sum = sum + pr_prod_mult(ts, pr_rc_num(ts), i)
    end do
    do i = 1, pr_num_arrangements(ts, pr_rc_num(ts))
      pr_prod_mult(ts, pr_rc_num(ts), i) = pr_prod_mult(ts, pr_rc_num(ts), i) / sum
    end do

    return
  end

```

Set up isotopic variants of ionizations. By convention, an ionization reaction specification takes the form:



note that B could be charged to start with.

```

⟨ Functions and Subroutines 1.2 ⟩ +≡
subroutine set_prod_ionize(pr_test_dummy(ts), sp_dummy(reagents), rc_dummy(r))
  implicit none f77
  ps_common
  rc_common
  pr_common
  sp_common
  implicit none f90

  pr_test_decl(ts)
  sp_decl(reagents_*)
  rc_decl(r)

  external species_add_check // External
  logical species_add_check

  integer i // Local
  sp_decl(product2)
  st_decls

  assert(rc_product_num(r) ≡ 3)
  pr_num_arrangements(ts, pr_rc_num(ts)) = 1
  pr_prod_mult(ts, pr_rc_num(ts), 1) = one

  /* First product corresponds to the reagent which does the ionizing: its species is unaltered. */
  assert(sp_generic(rc_product(r, 1)) ≡ sp_generic(reagents1))
  pr_ts_prod(ts, pr_rc_num(ts), 1, 1) = reagents1

  /* By convention, the third (last) product is the electron knocked off during ionization. */
  assert(rc_product(r, 3) ≡ sp_lookup('e'))
  pr_ts_prod(ts, pr_rc_num(ts), 1, 3) = rc_product(r, 3)

  /* The second product is the result of the ionization. It corresponds to the second reagent, minus
   an electron. By looping over the generic equivalents to the second rc_product, we ensure that any
   details (e.g., quantum state) not checked by species_add_check are correctly carried through. */
  product2 = rc_product(r, 3)
  do i = 1, num_equiv_genrics_rc_product(r, 2)
    product1 = equivalents_i, genrics_rc_product(r, 2)
    if (species_add_check(2, sp_args(product), 1, sp_args(reagents2))) then
      pr_ts_prod(ts, pr_rc_num(ts), 1, 2) = product1
      return
    end if
  end do
  assert('Unmatched_ionization_product' ≡ '_')
  return
end
```

Set up isotopic variants of charge exchange. By convention, a charge exchange reaction specification takes the form:



the superscript “+”s are used only to indicate a change in charge state; the actual charge states of A and B are intended to be arbitrary. The procedure below can handle both single and double electron charge exchange.

⟨ Functions and Subroutines 1.2 ⟩ +≡

```

subroutine set_prod_chargex(pr_test_dummy(ts), sp_dummy(reagents), rc_dummy(r))
  implicit none f77
  rc_common
  pr_common
  sp_common
  ps_common
  implicit none f90
  pr_test_decl(ts)
  rc_decl(r)
  sp_decl(reagents_*)
  external species_add_check // External
  logical species_add_check
  integer i, num_electrons // Local
  sp_decl(product_3)
  st_decls
  num_electrons = 0
  assert(rc_product_num(r) ≡ 2)
  pr_num_arrangements(ts, pr_rc_num(ts)) = 1
  pr_prod_mult(ts, pr_rc_num(ts), 1) = one
    /* The first product should match up with the first reagent plus one or two electrons. */
  product_1 = sp_lookup('e')
  product_2 = reagents_1
  do i = 1, num_equiv_genrics_rc_product(r, 1)
    if (species_add_check(2, sp_args(product), 1, sp_args(equivalents_i, genrics_rc_product(r, 1)))) then
      pr_ts_prod(ts, pr_rc_num(ts), 1, 1) = equivalents_i, genrics_rc_product(r, 1)
      num_electrons = 1
      goto loop1
    else // Might be double charge exchange
      product_3 = product_1
      if (species_add_check(3, sp_args(product), 1, sp_args(equivalents_i, genrics_rc_product(r, 1)))) then
        pr_ts_prod(ts, pr_rc_num(ts), 1, 1) = equivalents_i, genrics_rc_product(r, 1)
        num_electrons = 2
        goto loop1
      end if
    end if
  end do
  assert('Unmatched first charge exchange product' ≡ ' ')
loop1: continue
  /* Likewise, the second reagent should match the second product plus one or two electrons. */
  do i = 1, num_equiv_genrics_rc_product(r, 2)

```

```
product2 = equivalentsi, generics_rc_product(r, 2)
if ((num_electrons ≡ 1) ∧ (species_add_check(2, sp_args(product), 1, sp_args(reagents2)))) then
    pr_ts_prod(ts, pr_rc_num(ts), 1, 2) = product2
    return
else if (num_electrons ≡ 2) then
    if (species_add_check(3, sp_args(product), 1, sp_args(reagents2))) then
        pr_ts_prod(ts, pr_rc_num(ts), 1, 2) = product2
        return
    end if
end if
end do
assert('Unmatched_\u00b9second_\u00b9charge_\u00b9exchange_\u00b9product' ≡ '\u00b9')
return
end
```

Set up isotopic variants of elastic collisions. By convention, an elastic reaction specification takes the form:



the superscript “+”s are used only to indicate a charge state; the actual charge states of A and B are intended to be arbitrary. Currently the only reaction allowed is that given below.



*⟨ Functions and Subroutines 1.2 ⟩ +≡*

```

subroutine set_prod_elastic(pr_test_dummy(ts), sp_dummy(reagents), rc_dummy(r))
  implicit none_f77
  rc_common
  pr_common
  sp_common
  implicit none_f90
  pr_test_decl(ts)
  rc_decl(r)
  sp_decl(reagents_*)
  external species_add_check // External
  logical species_add_check
  st_decls
  assert(rc_product_num(r) ≡ 2)
  pr_num_arrangements(ts, pr_rc_num(ts)) = 1
  pr_prod_mult(ts, pr_rc_num(ts), 1) = one
  assert(sp_generic(rc_product(r, 1)) ≡ sp_generic(reagents_1))
  @#if 0
    /* Having tested reagents and stated assumptions in the introduction, why did we write this at
     all? */
    pr_ts_prod(ts, pr_rc_num(ts), 1, 1) = rc_product(r, 1)
  @#endif
  pr_ts_prod(ts, pr_rc_num(ts), 1, 1) = reagents_1
  assert(sp_generic(rc_product(r, 2)) ≡ sp_generic(reagents_2))
  @#if 0
    pr_ts_prod(ts, pr_rc_num(ts), 1, 2) = rc_product(r, 2)
  @#endif
  pr_ts_prod(ts, pr_rc_num(ts), 1, 2) = reagents_2
  return
end

```

Set up isotopic variants of dissociation. This reactions is assumed to have the form:



where the products  $B_j$  are the result of the dissociation of B.

```

⟨ Functions and Subroutines 1.2 ⟩ +≡
  subroutine set_prod_dissoc(pr_test_dummy(ts), sp_dummy(reagents), rc_dummy(r))
    implicit none_f77
    rc_common
    pr_common
    sp_common
    implicit none_f90
    pr_test_decl(ts)
    rc_decl(r)
    sp_decl(reagents_*)
    integer i    // Local
    assert(sp_generic(rc_product(r, 1)) ≡ sp_generic(reagents_1))
    if (rc_product_num(r) ≡ 3) then
      call product_perm_2(pr_test_args(ts), 1, sp_args(reagents_2), rc_args(r))
    else if (rc_product_num(r) ≡ 4) then
      call product_perm_3(pr_test_args(ts), 1, sp_args(reagents_2), rc_args(r))
    else
      assert('Illegal value of rc_product_num' ≡ ' ')
    end if
    do i = 1, pr_num_arrangements(ts, pr_rc_num(ts))
      pr_ts_prod(ts, pr_rc_num(ts), i, 1) = reagents_1
    end do
    return
  end

```

Set up isotopic variants of dissociative recombination. This reactions is assumed to have the form:



where the products  $C_j$  are the result of the dissociation of C which is formed by the recombination of A and B.

{Functions and Subroutines 1.2} +≡

```

subroutine set_prod_dissoc_rec(pr_test_dummy(ts), sp_dummy(reagents), rc_dummy(r))
  implicit none f77
  rc_common
  pr_common
  sp_common
  implicit none f90

  pr_test_decl(ts)
  rc_decl(r)
  sp_decl(reagents_*)

```

```

if (rc_product_num(r) ≡ 2) then
  call product_perm_2(pr_test_args(ts), 2, sp_args(reagents), rc_args(r))
else if (rc_product_num(r) ≡ 3) then
  call product_perm_3(pr_test_args(ts), 2, sp_args(reagents), rc_args(r))
else
  assert('Illegal value of rc_product_num' ≡ ' ')
end if

return
end

```

Set up isotopic variants of excitation. This routine is also suitable for deexcitation. The general form of these reactions is:



where A is the background species and B is the test species. The \* indicates a change in the electron configuration of the species. In terms of DEGAS 2's species characteristics, this is just a change in the species label; the elements comprising the reagent and product test species must be the same.

*< Functions and Subroutines 1.2 > +≡*

```

subroutine set_prod_excite(pr_test_dummy(ts), sp_dummy(reagents), rc_dummy(r))
  implicit none f77
  ps_common
  rc_common
  pr_common
  sp_common
  implicit none f90

  pr_test_decl(ts)
  sp_decl(reagents*)
  rc_decl(r)

  external species_add_check // External
  logical species_add_check

  integer i // Local
  sp_decl(product)
  st_decls

  assert(rc_product_num(r) ≡ 2)
  pr_num_arrangements(ts, pr_rc_num(ts)) = 1
  pr_prod_mult(ts, pr_rc_num(ts), 1) = one
  /* First product corresponds to the reagent which does the exciting: its species is unaltered. */
  assert(sp_generic(rc_product(r, 1)) ≡ sp_generic(reagents_1))
  pr_ts_prod(ts, pr_rc_num(ts), 1, 1) = reagents_1 /* The second product is the result of the
  excitation or deexcitation. It corresponds only to a different electron configuration from the
  reagent. So, species_add_check will suffice for picking out the appropriate product. */
  do i = 1, num_equiv_genrics_rc_product(r, 2)
    product = equivalents_i, genrics_rc_product(r, 2)
    if (species_add_check(1, sp_args(product), 1, sp_args(reagents_2))) then
      pr_ts_prod(ts, pr_rc_num(ts), 1, 2) = product
      return
    end if
  end do
  assert('Unmatched_excitation_product' ≡ '_')
  return
end

```

This is a clearing-house routine used to select the product-setting routine specific for each reaction among background species. Presently, the only application for this is recombination.

```

⟨ Functions and Subroutines 1.2 ⟩ +≡
subroutine set_bkrc_products(bkrc_ind, sp_dummy(reagents))
  implicit none_f77
  pr_common
  rc_common
  implicit none_f90

  integer bkrc_ind    // Input
  sp_decl(reagents_*)
  integer i      // Local

  do i = 1, rc_product_max
    pr_bkrc_prod(bkrc_ind, i) = int_unused
  end do
  if (rc_reaction_type(pr_reaction(pr_bk_rc(bkrc_ind))) ≡ 'recombination') then
    call set_prod_recombine(bkrc_ind, sp_args(reagents))
  else
    assert('Unsupported_reaction_type' ≡ '_')
    return
  end if
  return
end

```

Set up isotopic variants of recombinations. By convention, a recombination reaction specification takes the form:



note that B could be charged to start with.

*⟨ Functions and Subroutines 1.2 ⟩ +≡*

```

subroutine set_prod_recombine(bkrc_ind, sp_dummy(reagents))
  implicit none f77
  rc_common
  pr_common
  sp_common
  ps_common
  implicit none f90

  integer bkrc_ind
  sp_decl(reagents_*)
  external species_add_check // External
  logical species_add_check

  integer i // Local
  rc_decl(r)
  sp_decl(product_2)
  st_decls

  r = pr_reaction(pr_bk_rc(bkrc_ind))
  assert(rc_product_num(r) ≡ 1)

  /* Assuming that only one product is listed for recombination. If there are more products, not
   only does this routine have to be augmented, but will probably need to add common arrays
   to deal with product permutations. By looping over the generic equivalents to rc_product, we
   ensure that any details (e.g., quantum state) not checked by species_add_check are correctly
   carried through. */

  do i = 1, num_equiv_generics_rc_product(r, 1)
    product_1 = equivalents_i, generics_rc_product(r, 1)
    if (species_add_check(1, sp_args(product), 2, sp_args(reagents))) then
      pr_bkrc_prod(bkrc_ind, 1) = product_1
      return
    end if
  end do
  assert('Unmatched_recombination_product' ≡ ' ')
  return
end

```

This is a clearing-house routine used to select the product-setting routine specific for each PMI.

```

⟨ Functions and Subroutines 1.2 ⟩ +≡
subroutine set_pmi_products(pr_test_dummy(ts), pm_dummy(p))
  implicit none f77
  pr_common
  pm_common
  implicit none f90

  pr_test_decl(ts) // Input
  pm_decl(p)

  integer i // Local
  real sum

  if (pm_pmi_type(p) ≡ 'reflection') then
    call set_prod_reflection(pr_test_args(ts), pm_args(p))
  else if (pm_pmi_type(p) ≡ 'adsorption') then
    call set_prod_adsorption(pr_test_args(ts), pm_args(p))
  else if (pm_pmi_type(p) ≡ 'desorption') then
    call set_prod_desorption(pr_test_args(ts), pm_args(p))
  else if (pm_pmi_type(p) ≡ 'sputter') then
    assert('set_product_for_sputter_not_available' ≡ ' ')
  end if

  /* Rescale multiplicity so that it becomes a probability */
  sum = zero
  do i = 1, pr_pm_num_arrange(ts, pr_pm_case_num(ts))
    sum = sum + pr_pm_prod_mult(ts, pr_pm_case_num(ts), i)
  end do
  if (sum > zero) then // In some cases, pr_pm_prod_mult is ≡ 0.
    do i = 1, pr_pm_num_arrange(ts, pr_pm_case_num(ts))
      pr_pm_prod_mult(ts, pr_pm_case_num(ts), i) = pr_pm_prod_mult(ts, pr_pm_case_num(ts), i)/sum
    end do
  end if

  return
end

```

Set up isotopic variations of reflections. This is a simple case of “what comes in must go out.”

```
<Functions and Subroutines 1.2> +≡
subroutine set_prod_reflection(pr_test_dummy(ts), pm_dummy(p))
  implicit none_f77
  pr_common // Common
  pm_common
  sp_common
  implicit none_f90

  pr_test_decl(ts) // Input
  pm_decl(p)

  assert(pm_product_num(p) ≡ 1)
  pr_pm_num_arrange(ts, pr_pm_case_num(ts)) = 1
  pr_pm_prod_mult(ts, pr_pm_case_num(ts), 1) = one

  /* The only product had better be the same as the incoming test particle */
  assert(sp_generic(pm_product(p, 1)) ≡ sp_generic(pr_test(ts)))
  pr_pm_prod(ts, pr_pm_case_num(ts), 1, 1) = pr_test(ts)

return
end
```

Set up isotopic variations for adsorptions. It doesn’t get any easier than this.

```
<Functions and Subroutines 1.2> +≡
subroutine set_prod_adsorption(pr_test_dummy(ts), pm_dummy(p))
  implicit none_f77
  pr_common // Common
  pm_common
  implicit none_f90

  pr_test_decl(ts) // Input
  pm_decl(p)

  assert(pm_product_num(p) ≡ 0)
  pr_pm_num_arrange(ts, pr_pm_case_num(ts)) = 1
  pr_pm_prod_mult(ts, pr_pm_case_num(ts), 1) = one

  pr_pm_prod(ts, pr_pm_case_num(ts), 1, 1) = 0

return
end
```

Set up isotopic variations for desorptions. The assumptions made here reflect the algorithm used in the original DEGAS code. Namely, for most incident neutral species, the desorbed species is the same as the incident one. For an incident hydrogen atom, however, the possibility existed for desorption as a molecule. It is intended that this option be represented by two products in the reference description of the PMI, only one of which is used at a time here. In the event that a molecule is returned, the isotope of one of the atoms is required to match that of the incident H. The other atom will be assigned in a general way here based on the other isotopes present in the problem. It is assumed that the weights of these various options are established in the particular data file for this PMI and will be implemented only at run time. An analogous description holds for carbon as well as the various methane fragments.

```

⟨ Functions and Subroutines 1.2 ⟩ +≡
subroutine set_prod_desorption(pr_test_dummy(ts), pm_dummy(p))
  implicit none_f77
  pr_common // Common
  pm_common
  el_common
  ps_common
  implicit none_f90

  pr_test_decl(ts) // Input
  pm_decl(p)

  external species_el_count // External
  integer species_el_count

  integer i, ic, j, count_ts, count_prod // Local
  logical match

  pr_generic_decl(generic_prod)

  pr_pm_num_arrange(ts, pr_pm_case_num(ts)) = 0
  do i = 1, pm_product_num(p)
    generic_prod = genericpm_product(p, i)
    assert(generic_prod > 0  $\wedge$  generic_prod  $\leq$  num_generics)
    do j = 1, num_equivgeneric_prod
      match =  $\mathcal{T}$ 
      do ic = 1, el_num
        count_ts = species_el_count(pr_test(ts), el_args(ic))
        count_prod = species_el_count(equivalentsj, generic_prod, el_args(ic))
        if (count_ts > count_prod)
          match =  $\mathcal{F}$ 
      end do
      if (match) then
        pr_pm_num_arrange(ts, pr_pm_case_num(ts))++
        /* Set this to zero; it will be determined at run time. */
        pr_pm_prod_mult(ts, pr_pm_case_num(ts), pr_pm_num_arrange(ts,
          pr_pm_case_num(ts))) = zero
        pr_pm_prod(ts, pr_pm_case_num(ts), pr_pm_num_arrange(ts, pr_pm_case_num(ts)),
          1) = equivalentsj, generic_prod
      end if
    end do
  end do
  assert(pr_pm_num_arrange(ts, pr_pm_case_num(ts)) > 0)

  return
end

```

Assemble and check permutations of two products arising from the dissociation of  $num\_sp$  species  $sp$  in reaction  $r$ . The test species  $ts$  is carried into the routine since to be used as an array index. Note that for simplicity duplicates (i.e., permuted lists of equivalent species) are not eliminated. This makes for more arrangements than necessary, but the overall probability for a given unique arrangement is effected nonetheless. If desired, this duplication could be eliminated through additional logic.

```

⟨ Functions and Subroutines 1.2 ⟩ +≡
subroutine product_perm_2(pr_test_dummy(ts), num_sp, sp_dummy(sp), rc_dummy(r))
  implicit none_f77
  rc_common
  sp_common
  pr_common
  ps_common
  implicit none_f90

  integer num_sp    // Input
  pr_test_decl(ts)
  sp_decl(sp_num_sp)
  rc_decl(r)

  external species_add_check // External
  logical species_add_check

  integer i, j, first_product // Local
  sp_decl(product_2)
  pr_generic_decl(generic_1)
  pr_generic_decl(generic_2)

  /* A num_sp of 1 indicates pure dissociation ⇒ the actual first product will be the dissociating
   agent; the first dissociation product is then the second product. On the other hand, num_sp =
   2 suggests dissociative recombination where all products come from dissociation. */
  if (num_sp ≡ 1) then
    first_product = 2
  else if (num_sp ≡ 2) then
    first_product = 1
  else
    assert('Invalid_num_sp' ≡ '_')
  end if

  /* Obtain generic indices of the two products */
  generic_1 = generic_rc_product(r, first_product)
  generic_2 = generic_rc_product(r, first_product+1)

  /* Loop over the species equivalent to each generic and see which combinations are consistent
   with the given reagents. */
  pr_num_arrangements(ts, pr_rc_num(ts)) = 0
  do i = 1, num_equiv_generic_1
    product_1 = equivalents_i, generic_1
    do j = 1, num_equiv_generic_2
      product_2 = equivalents_j, generic_2
      if (species_add_check(2, sp_args(product), num_sp, sp_args(sp))) then
        pr_num_arrangements(ts, pr_rc_num(ts))++ // Valid arrangement
        assert(pr_num_arrangements(ts, pr_rc_num(ts)) ≤ pr_arrangement_max)
        pr_ts_prod(ts, pr_rc_num(ts), pr_num_arrangements(ts, pr_rc_num(ts)),
                   first_product) = product_1
      end if
    end do
  end do
end subroutine

```

```
pr_ts_prod(ts, pr_rc_num(ts), pr_num_arrangements(ts, pr_rc_num(ts)),
           first_product + 1) = product2
pr_prod_mult(ts, pr_rc_num(ts), pr_num_arrangements(ts,
           pr_rc_num(ts))) = sp_multiplicity(product1) * sp_multiplicity(product2)
end if
end do
end do
assert(pr_num_arrangements(ts, pr_rc_num(ts)) > 0)
return
end
```

Assemble and check permutations of three products arising from the dissociation of species  $sp$  in reaction  $r$ . The test species  $ts$  is carried into the routine since to be used as an array index. Note that for simplicity duplicates (i.e., permuted lists of equivalent species) are not eliminated. This makes for more arrangements than necessary, but the overall probability for a given unique arrangement is effected nonetheless. If desired, this duplication could be eliminated through additional logic.

```

⟨ Functions and Subroutines 1.2 ⟩ +≡
subroutine product_perm_3(pr_test_dummy(ts), num_sp, sp_dummy(sp), rc_dummy(r))
  implicit none_f77
  rc_common
  sp_common
  pr_common
  ps_common
  implicit none_f90

  integer num_sp    // Input
  pr_test_decl(ts)
  sp_decl(sp_num_sp)
  rc_decl(r)

  external species_add_check // External
  logical species_add_check

  integer i, j, k, first_product // Local
  sp_decl(product_3)
  pr_generic_decl(generic_1)
  pr_generic_decl(generic_2)
  pr_generic_decl(generic_3)

  /* A num_sp of 1 indicates pure dissociation ⇒ the actual first product will be the dissociating
   agent; the first dissociation product is then the second product. On the other hand, num_sp =
   2 suggests dissociative recombination where all products come from dissociation. */
  if (num_sp ≡ 1) then
    first_product = 2
  else if (num_sp ≡ 2) then
    first_product = 1
  else
    assert('Invalid_num_sp' ≡ '_')
  end if

  /* Obtain generic indices for each product */
  generic_1 = generic_rc_product(r, first_product)
  generic_2 = generic_rc_product(r, first_product+1)
  generic_3 = generic_rc_product(r, first_product+2)

  /* Loop over the species equivalent to each generic and see which combinations are consistent
   with the given reagents. */
  pr_num_arrangements(ts, pr_rc_num(ts)) = 0
  do i = 1, num_equiv_generic_1
    product_1 = equivalents_i, generic_1
    do j = 1, num_equiv_generic_2
      product_2 = equivalents_j, generic_2
      do k = 1, num_equiv_generic_3
        product_3 = equivalents_k, generic_3
        if (species_add_check(3, sp_args(product), num_sp, sp_args(sp))) then

```

```
pr_num_arrangements(ts, pr_rc_num(ts))++ // Valid arrangement
assert(pr_num_arrangements(ts, pr_rc_num(ts)) ≤ pr_arrangement_max)
pr_ts_prod(ts, pr_rc_num(ts), pr_num_arrangements(ts, pr_rc_num(ts)),
           first_product) = product1
pr_ts_prod(ts, pr_rc_num(ts), pr_num_arrangements(ts, pr_rc_num(ts)),
           first_product + 1) = product2
pr_ts_prod(ts, pr_rc_num(ts), pr_num_arrangements(ts, pr_rc_num(ts)),
           first_product + 2) = product3
pr_prod_mult(ts, pr_rc_num(ts), pr_num_arrangements(ts, pr_rc_num(ts))) =
    sp_multiplicity(product1) * sp_multiplicity(product2) * sp_multiplicity(product3)
end if
end do
end do
end do
assert(pr_num_arrangements(ts, pr_rc_num(ts)) > 0)
return
end
```

## 2 References

### 3 INDEX

*assert*: 1.2, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 1.10, 1.11, 1.13, 1.15, 1.16, 1.17, 1.18, 1.19, 1.20, 1.21, 1.22, 1.23, 1.24, 1.25, 1.26, 1.27, 1.28.  
*b*: 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 1.10.  
*back*: 1.13.  
*background*: 1.13.  
*bk\_reagent*: 1.13.  
*bkrc\_ind*: 1.21, 1.22.  
*char\_undef*: 1.2.  
*char\_uninit*: 1.11.  
*check\_problem*: 1.2, 1.13.  
*count\_prod*: 1.26.  
*count\_ts*: 1.26.  
*datasetup*: 1.  
*diskin*: 1.2.  
*e*: 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 1.10.  
*el\_args*: 1.26.  
*el\_common*: 1.26.  
*el\_num*: 1.26.  
*eof*: 1.2, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 1.10.  
*equivalents*: 1.13, 1.15, 1.16, 1.20, 1.22, 1.26, 1.27, 1.28.  
*file*: 1.2.  
*FILE*: 1.  
*FILELEN*: 1.2.  
*filenames\_array*: 1.2.  
*finish\_var0\_list*: 1.1, 1.12.  
*first\_product*: 1.27, 1.28.  
*float\_abort*: 1.1.  
*form*: 1.2.  
*FORTRAN77*: 1.1.  
*generic*: 1.13.  
*generic\_prod*: 1.26.  
*generic\_reagents*: 1.13.  
*generic\_1*: 1.27, 1.28.  
*generic\_2*: 1.27, 1.28.  
*generic\_3*: 1.28.  
*generics*: 1.3, 1.4, 1.5, 1.13, 1.15, 1.16, 1.20, 1.22, 1.26, 1.27, 1.28.  
*hrecombine*: 1.13.  
*i*: 1.3, 1.4, 1.5, 1.7, 1.8, 1.9, 1.11, 1.12, 1.13, 1.14, 1.15, 1.16, 1.18, 1.20, 1.21, 1.22, 1.23, 1.26, 1.27, 1.28.  
*i\_e\_rate*: 1.12.  
*ic*: 1.26.  
*implicit\_none\_f77*: 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 1.10, 1.11, 1.12, 1.13, 1.14, 1.15, 1.16, 1.17, 1.18, 1.19, 1.20, 1.21, 1.22, 1.23, 1.24, 1.25, 1.26, 1.27, 1.28.  
*implicit\_none\_f90*: 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 1.10, 1.11, 1.12, 1.13, 1.14, 1.15, 1.16, 1.17, 1.18, 1.19, 1.20, 1.21, 1.22, 1.23, 1.24, 1.25, 1.26, 1.27, 1.28.  
*init\_pmi*: 1.1.  
*init\_problem*: 1.2, 1.3.  
*init\_reaction*: 1.1.  
*init\_var0\_list*: 1.1, 1.11.  
*int\_lookup*: 1.13.  
*int\_unused*: 1.13, 1.21.  
*j*: 1.13, 1.26, 1.27, 1.28.  
*k*: 1.13, 1.28.  
*l*: 1.13.  
*len*: 1.2, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 1.10.  
*length*: 1.2, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 1.10.  
*line*: 1.2, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 1.10, 1.12.  
*LINE*: 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 1.10.  
*LINELEN*: 1.2.  
*lines*: 1.12.  
*loop1*: 1.2, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 1.10, 1.16.  
*loop2*: 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 1.10.  
*m*: 1.13.  
*ma\_common*: 1.7, 1.13.  
*ma\_lookup*: 1.7.  
*ma\_name*: 1.13.  
*ma\_num*: 1.7.  
*match*: 1.26.  
*max*: 1.13.  
*mem\_alloc*: 1.11.  
*mem\_size*: 1.11.  
*nc\_read\_elements*: 1.3.  
*nc\_read\_materials*: 1.3.  
*nc\_read\_pmi*: 1.3.  
*nc\_read\_reactions*: 1.3.  
*nc\_read\_species*: 1.3.  
*nc\_write\_problem*: 1.1.  
*next\_token*: 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 1.10.  
*num\_electrons*: 1.16.  
*num\_equiv*: 1.13, 1.15, 1.16, 1.20, 1.22, 1.26, 1.27, 1.28.  
*num\_generics*: 1.3, 1.4, 1.5, 1.13, 1.26.  
*num\_lines*: 1.12.  
*num\_sp*: 1.27, 1.28.  
*old\_num*: 1.12.  
*one*: 1.13, 1.15, 1.16, 1.17, 1.20, 1.24, 1.25.

*p*: 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 1.10.  
*parse\_string*: 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 1.10.  
*pm\_args*: 1.13, 1.23.  
*pm\_common*: 1.8, 1.13, 1.23, 1.24, 1.25, 1.26.  
*pm\_decl*: 1.23, 1.24, 1.25, 1.26.  
*pm\_dummy*: 1.23, 1.24, 1.25, 1.26.  
*pm\_gen*: 1.13.  
*pm\_lookup*: 1.8.  
*pm\_name*: 1.13.  
*pm\_num*: 1.8.  
*pm\_pmi\_type*: 1.23.  
*pm\_product*: 1.13, 1.24, 1.26.  
*pm\_product\_num*: 1.13, 1.24, 1.25, 1.26.  
*pm\_reagent*: 1.13.  
*pmi\_generic\_no*: 1.13.  
*pmi\_generic\_yes*: 1.13.  
*pr*: 1.13.  
*pr\_arrangement\_max*: 1.13, 1.27, 1.28.  
*pr\_background*: 1.5, 1.11, 1.13.  
*pr\_background\_check*: 1.13.  
*pr\_background\_decl*: 1.13.  
*pr\_background\_lookup*: 1.13.  
*pr\_background\_num*: 1.3, 1.5, 1.11, 1.13.  
*pr\_bk\_rc*: 1.13, 1.21, 1.22.  
*pr\_bkrc\_dim*: 1.13.  
*pr\_bkrc\_num*: 1.13.  
*pr\_bkrc\_prod*: 1.13, 1.21, 1.22.  
*pr\_bkrc\_reagent\_max*: 1.13.  
*pr\_bkrc\_rg*: 1.13.  
*pr\_common*: 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9,  
 1.10, 1.11, 1.12, 1.13, 1.14, 1.15, 1.16, 1.17, 1.18,  
 1.19, 1.20, 1.21, 1.22, 1.23, 1.24, 1.25, 1.26, 1.27,  
 1.28.  
*pr\_ex\_rc*: 1.10, 1.13.  
*pr\_ex\_rc\_prod*: 1.13.  
*pr\_ex\_test*: 1.9, 1.13.  
*pr\_ex\_test\_check*: 1.13.  
*pr\_ex\_test\_dim*: 1.9, 1.13.  
*pr\_ex\_test\_lookup*: 1.13.  
*pr\_ex\_test\_num*: 1.3, 1.9, 1.13.  
*pr\_ex\_ts\_bk*: 1.13.  
*pr\_ex\_ts\_rc*: 1.13.  
*pr\_ex\_ts\_rc\_num*: 1.13.  
*pr\_exrc\_dim*: 1.10, 1.13.  
*pr\_exrc\_num*: 1.3, 1.10, 1.13.  
*pr\_generic\_decl*: 1.13, 1.26, 1.27, 1.28.  
*pr\_mat\_ref*: 1.7, 1.13.  
*pr\_materials\_num*: 1.3, 1.7, 1.13.  
*pr\_max\_equiv*: 1.13.  
*pr\_max\_lines*: 1.12.  
*pr\_num\_arrangements*: 1.13, 1.14, 1.15, 1.16, 1.17,  
 1.18, 1.20, 1.27, 1.28.  
*pr\_num\_change\_vars*: 1.11.  
*pr\_pm\_case\_num*: 1.13, 1.23, 1.24, 1.25, 1.26.  
*pr\_pm\_cases*: 1.13.  
*pr\_pm\_num\_arrange*: 1.13, 1.23, 1.24, 1.25, 1.26.  
*pr\_pm\_prod*: 1.13, 1.24, 1.25, 1.26.  
*pr\_pm\_prod\_mult*: 1.13, 1.23, 1.24, 1.25, 1.26.  
*pr\_pm\_ref*: 1.8, 1.13.  
*pr\_pmi\_max*: 1.13.  
*pr\_pmi\_num*: 1.3, 1.8, 1.13.  
*pr\_problem\_sp\_back*: 1.11.  
*pr\_problem\_sp\_test*: 1.11.  
*pr\_prod\_mult*: 1.13, 1.14, 1.15, 1.16, 1.17, 1.20, 1.27,  
 1.28.  
*pr\_rc\_num*: 1.13, 1.14, 1.15, 1.16, 1.17, 1.18, 1.20,  
 1.27, 1.28.  
*pr\_reaction*: 1.6, 1.10, 1.13, 1.21, 1.22.  
*pr\_reaction\_dim*: 1.6, 1.10.  
*pr\_reaction\_max*: 1.13.  
*pr\_reaction\_num*: 1.3, 1.6, 1.10, 1.13.  
*pr\_tag\_string\_length*: 1.12.  
*pr\_test*: 1.4, 1.11, 1.13, 1.24, 1.26.  
*pr\_test\_args*: 1.13, 1.14, 1.18, 1.19, 1.23.  
*pr\_test\_check*: 1.13.  
*pr\_test\_decl*: 1.13, 1.14, 1.15, 1.16, 1.17, 1.18, 1.19,  
 1.20, 1.23, 1.24, 1.25, 1.26, 1.27, 1.28.  
*pr\_test\_dummy*: 1.14, 1.15, 1.16, 1.17, 1.18, 1.19,  
 1.20, 1.23, 1.24, 1.25, 1.26, 1.27, 1.28.  
*pr\_test\_lookup*: 1.13.  
*pr\_test\_num*: 1.3, 1.4, 1.11, 1.13.  
*pr\_ts\_bk*: 1.13.  
*pr\_ts\_prod*: 1.13, 1.15, 1.16, 1.17, 1.18, 1.20, 1.27,  
 1.28.  
*pr\_ts\_rc*: 1.13.  
*pr\_var\_angle*: 1.11.  
*pr\_var\_emitter\_v\_vector*: 1.11.  
*pr\_var\_emitter\_v\_2*: 1.11.  
*pr\_var\_emitter\_v\_3*: 1.11.  
*pr\_var\_emitter\_vf\_Maxwell\_vector*: 1.11.  
*pr\_var\_emitter\_vf\_Maxwell\_2*: 1.11.  
*pr\_var\_emitter\_vf\_Maxwell\_3*: 1.11.  
*pr\_var\_emitter\_vth\_Maxwell*: 1.11.  
*pr\_var\_energy*: 1.11.  
*pr\_var\_energy\_change*: 1.11.  
*pr\_var\_energy\_in*: 1.11.  
*pr\_var\_energy\_out*: 1.11.  
*pr\_var\_mass*: 1.11.  
*pr\_var\_mass\_change*: 1.11.  
*pr\_var\_mass\_in*: 1.11.  
*pr\_var\_mass\_out*: 1.11.  
*pr\_var\_momentum\_change\_vector*: 1.11.  
*pr\_var\_momentum\_change\_2*: 1.11.  
*pr\_var\_momentum\_change\_3*: 1.11.

*pr\_var\_momentum\_in\_vector*: 1.11.  
*pr\_var\_momentum\_in\_2*: 1.11.  
*pr\_var\_momentum\_in\_3*: 1.11.  
*pr\_var\_momentum\_out\_vector*: 1.11.  
*pr\_var\_momentum\_out\_2*: 1.11.  
*pr\_var\_momentum\_out\_3*: 1.11.  
*pr\_var\_momentum\_vector*: 1.11.  
*pr\_var\_momentum\_2*: 1.11.  
*pr\_var\_momentum\_3*: 1.11.  
*pr\_var\_problem\_sp\_index*: 1.11.  
*pr\_var\_unknown*: 1.11.  
*pr\_var\_xz\_stress*: 1.11.  
*pr\_var0\_list*: 1.11, 1.12.  
*pr\_var0\_num*: 1.11, 1.12.  
*problem\_background\_reaction*: 1.13.  
*problem\_background\_sp*: 1.5, 1.13.  
*problem\_bkrc\_products*: 1.13.  
*problem\_bkrc\_reagents*: 1.13.  
*problem\_ex\_test\_sp*: 1.9, 1.13.  
*problem\_exrc\_products*: 1.13.  
*problem\_external\_reaction*: 1.10, 1.13.  
*problem\_external\_test\_background*: 1.13.  
*problem\_external\_test\_reaction*: 1.13.  
*problem\_external\_test\_reaction\_num*: 1.13.  
*problem\_infile*: 1.2.  
*problem\_materials\_ref*: 1.7, 1.13.  
*problem\_materials\_sub*: 1.7.  
*problem\_num\_arrangements*: 1.13.  
*problem\_pmi\_case\_num*: 1.13.  
*problem\_pmi\_cases*: 1.13.  
*problem\_pmi\_num\_arrange*: 1.13.  
*problem\_pmi\_prod\_mult*: 1.13.  
*problem\_pmi\_products*: 1.13.  
*problem\_pmi\_ref*: 1.8, 1.13.  
*problem\_pmi\_sub*: 1.8.  
*problem\_prod\_mult*: 1.13.  
*problem\_rc*: 1.6, 1.10, 1.13.  
*problem\_reaction\_num*: 1.13.  
*problem\_species\_background*: 1.5.  
*problem\_species\_test*: 1.4.  
*problem\_test\_background*: 1.13.  
*problem\_test\_products*: 1.13.  
*problem\_test\_reaction*: 1.13.  
*problem\_test\_sp*: 1.4, 1.13.  
*problem\_version*: 1.2.  
*problemfile*: 1.  
*problemsetup*: 1, 1.1.  
*product*: 1.15, 1.16, 1.20, 1.22, 1.27, 1.28.  
*product\_perm\_2*: 1.18, 1.19, 1.27.  
*product\_perm\_3*: 1.18, 1.19, 1.28.  
*ps\_common*: 1.3, 1.4, 1.5, 1.13, 1.15, 1.16, 1.20, 1.22, 1.26, 1.27, 1.28.  
*rc\_args*: 1.13, 1.14, 1.18, 1.19.  
*rc\_common*: 1.6, 1.10, 1.13, 1.14, 1.15, 1.16, 1.17, 1.18, 1.19, 1.20, 1.21, 1.22, 1.23, 1.24, 1.25, 1.26, 1.27, 1.28.  
*rc\_decl*: 1.14, 1.15, 1.16, 1.17, 1.18, 1.19, 1.20, 1.22, 1.23, 1.24, 1.25, 1.26, 1.27, 1.28.  
*rc\_dummy*: 1.14, 1.15, 1.16, 1.17, 1.18, 1.19, 1.20, 1.21, 1.22, 1.23, 1.24, 1.25, 1.26, 1.27, 1.28.  
*rc\_gen*: 1.13.  
*rc\_generic\_no*: 1.13.  
*rc\_generic\_yes*: 1.13.  
*rc\_lookup*: 1.6, 1.10.  
*rc\_name*: 1.13.  
*rc\_product*: 1.13, 1.15, 1.16, 1.17, 1.18, 1.20, 1.22, 1.23, 1.24, 1.25, 1.26, 1.27, 1.28.  
*rc\_product\_max*: 1.13, 1.21.  
*rc\_product\_num*: 1.13, 1.15, 1.16, 1.17, 1.18, 1.19, 1.20, 1.22.  
*rc\_reaction\_type*: 1.14, 1.21.  
*rc\_reagent*: 1.13.  
*rc\_reagent\_max*: 1.13.  
*rc\_reagent\_num*: 1.13.  
*read\_background*: 1.2, 1.5.  
*read\_ex\_reaction*: 1.2, 1.10.  
*read\_ex\_test*: 1.2, 1.9.  
*read\_materials*: 1.2, 1.7.  
*read\_pmi*: 1.2, 1.8.  
*read\_problem*: 1.1, 1.2.  
*read\_reaction*: 1.2, 1.6.  
*read\_string*: 1.2, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 1.10.  
*read\_test*: 1.2, 1.4.  
*readfilenames*: 1.1.  
*reagents*: 1.13, 1.14, 1.15, 1.16, 1.17, 1.18, 1.19, 1.20, 1.21, 1.22.  
*real\_unused*: 1.13.  
*rf\_common*: 1.2.  
*set\_bkrc\_products*: 1.13, 1.21.  
*set\_pmi\_products*: 1.13, 1.23.  
*set\_prod\_adsorption*: 1.23, 1.25.  
*set\_prod\_chargex*: 1.14, 1.16.  
*set\_prod\_desorption*: 1.23, 1.26.  
*set\_prod\_dissoc*: 1.14, 1.18.  
*set\_prod\_dissoc\_rec*: 1.14, 1.19.  
*set\_prod\_elastic*: 1.14, 1.17.  
*set\_prod\_excite*: 1.14, 1.20.  
*set\_prod\_ionize*: 1.14, 1.15.  
*set\_prod\_recombine*: 1.21, 1.22.  
*set\_prod\_reflection*: 1.23, 1.24.  
*set\_products*: 1.13, 1.14.  
*sp*: 1.27, 1.28.  
*SP*: 1.12.  
*sp\_args*: 1.13, 1.14, 1.15, 1.16, 1.18, 1.19, 1.20, 1.21, 1.22, 1.27, 1.28.

*sp\_common*: 1.3, 1.4, 1.5, 1.9, 1.11, 1.13, 1.15, 1.16, 1.17, 1.18, 1.19, 1.20, 1.22, 1.24, 1.27, 1.28.  
*sp\_decl*: 1.13, 1.14, 1.15, 1.16, 1.17, 1.18, 1.19, 1.20, 1.21, 1.22, 1.27, 1.28.  
*sp\_dummy*: 1.14, 1.15, 1.16, 1.17, 1.18, 1.19, 1.20, 1.21, 1.22, 1.27, 1.28.  
*sp\_generic*: 1.4, 1.5, 1.15, 1.17, 1.18, 1.20, 1.24.  
*sp\_lookup*: 1.4, 1.5, 1.9, 1.15, 1.16.  
*sp\_m*: 1.13.  
*sp\_multiplicity*: 1.27, 1.28.  
*sp\_name*: 1.13.  
*sp\_num*: 1.3, 1.4, 1.5.  
*sp\_sy*: 1.11.  
*species\_add\_check*: 1.15, 1.16, 1.17, 1.20, 1.22, 1.27, 1.28.  
*species\_el\_count*: 1.26.  
*st\_decls*: 1.2, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 1.10, 1.12, 1.13, 1.15, 1.16, 1.17, 1.20, 1.22.  
*status*: 1.2.  
*stdout*: 1.13.  
*string\_lookup*: 1.12.  
*sum*: 1.14, 1.23.  
*SUN*: 1.1.  
*tempfile*: 1.2.  
*test*: 1.13.  
*trim*: 1.13.  
*ts*: 1.14, 1.15, 1.16, 1.17, 1.18, 1.19, 1.20, 1.23, 1.24, 1.25, 1.26, 1.27, 1.28.  
*ts\_reagent*: 1.13.  
*unit*: 1.2, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 1.10.  
*var\_alloc*: 1.3, 1.4, 1.5, 1.7, 1.8, 1.11, 1.13.  
*var\_realloc*: 1.11, 1.12.  
*var\_realloca*: 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 1.10, 1.11, 1.12, 1.13.  
*var\_reallocb*: 1.12, 1.13.  
*xs\_copy*: 1.11, 1.12.  
*zero*: 1.14, 1.23, 1.26.

⟨ Functions and Subroutines 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 1.10, 1.11, 1.12, 1.13, 1.14, 1.15, 1.16, 1.17, 1.18, 1.19, 1.20, 1.21, 1.22, 1.23, 1.24, 1.25, 1.26, 1.27, 1.28 ⟩ Used in section 1.1.  
⟨ Memory allocation interface 0 ⟩ Used in sections 1.13, 1.12, 1.11, 1.10, 1.9, 1.8, 1.7, 1.6, 1.5, 1.4, 1.3, and 1.2.

**COMMAND LINE:** "fweave -f -i! -W[ -ybs15000 -ykw800 -ytw40000 -j -n/ /Users/dstotler/degas2/src/problemsetup.web".

**WEB FILE:** "/Users/dstotler/degas2/src/problemsetup.web".

**CHANGE FILE:** (none).

**GLOBAL LANGUAGE:** FORTRAN.